Arrays and I-structures

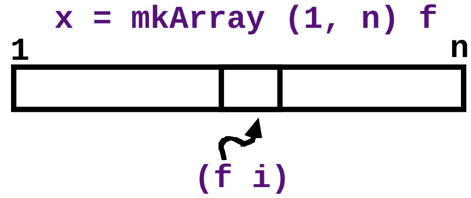
Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

October 12, 2006

Arrays

Cache for function values on a regular subdomain

`x = mkArray (1, n) f`

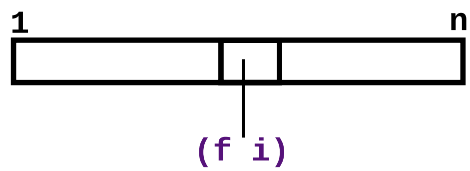


means **`x!i = (f i)`**
$1 \leq i \leq n$

Selection: **`x!i`** returns the value of the ith slot

Bounds: **`(bounds x)`** returns the tuple containing the bounds

Efficiency is the Motivation for Arrays



(f i) is computed once and stored

x!i is simply a fetch of a precomputed value and should take constant time

Index Type Class

Arrays can be indexed by any type that can be regarded as having a contiguous enumerable range

```
class Ix a where
   range    :: (a,a) -> [a]
   index    :: (a,a) -> a -> Int
   inRange  :: (a,a) -> a -> Bool
```

range: Returns the list of *index* elements between a lower and an upper bound

index: Given a *range* and an *index*, it returns an integer specifying the position of the index in the range based on 0

inRange : Tests if an *index* is in the *range*

Examples of Index Type

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

An index function may be defined as follows:

```
index (Sun,Sat) Wed = 3
index (Sun,Sat) Sat = 6
   ...
```

A two dimentional space may be indexed as followed:

```
index ((li,lj), (ui,uj)) (i,j) =
               (i-li)*((uj-lj)+1) + j - lj
```

This indexing function enumerates the space in the *row major* order

Array: An Abstract Datatype

```
module Array (Array, mkArray, (!), bounds)
   where

   infix 9 (!)

   data (Ix a) => Array a t
   mkArray  :: (Ix a) => (a,a) -> (a -> t) ->
                                   (Array a t)
   (!)      :: (Ix a) => (Array a t) -> a -> t
    bounds  :: (Ix a) => (Array a t) -> (a,a)
```

Thus,

```
type ArrayI  t = Array Int t
type MatrixI t = Array (Int,Int) t
```

Higher Dimensional Arrays

```
x = mkArray ((l1,l2),(u1,u2)) f
```

means `x!(i,j) = f (i,j)` $l1 \le i \le u1$
$l2 \le j \le u2$

Type

```
x  :: (Array (Int,Int) t)
```

Assuming

```
f :: (Int,Int) -> t
```

`mkArray` will work for higher dimensional matrices as well.

Array of Arrays

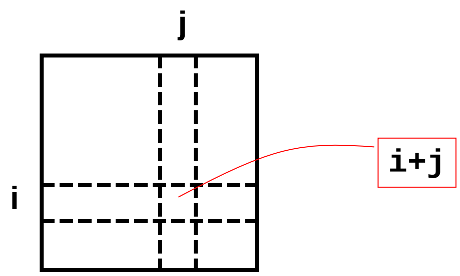
`(Array Int (Array Int t))` $\not\equiv$
`(Array (Int,Int) t)`

This allows flexibility in the implementation of higher dimensional arrays.

Matrices

```
add  (i,j) =  i + j

mkArray  ((1,1),(n,n))  add      ?
```



Transpose

```
transpose a =
     let
        ((l1,l2),(u1,u2)) = bounds a

        f (i,j) =  a!(j,i)           ?

     in
        mkArray  ((l2,l1),(u2,u1))  f
```

The *Wavefront* Example

$$x_{i,j} = x_{i-1,j} + x_{i,j-1}$$

1	1	1	1	1	1	1	1
1							
1							
1							
1							
1							
1							
1							

```
x = mkArray ((1,1),(n,n)) (f x)

f x (i, j) = if  i == 1 then 1
             else if  j == l  then  1
                  else x!(i-1,j) + x!(i,j-1)
```

fix point

Compute the least fix point.

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥

Wave-front parallelism

1	1	1	1	1	1	1	1
1	2	3	4	5	⊥	⊥	⊥
1	3	6	10	⊥	⊥	⊥	⊥
1	4	10	⊥	⊥	⊥	⊥	⊥
1	5	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥

```
x = mkArray ((1,1),(n,n)) (f x)

f x (i, j) = if  i == 1 then 1
             else if  j == l  then  1
                  else x!(i-1,j) + x!(i,j-1)
```

Array Comprehension

A special function to turn a list of (index,value) pairs into an array

```
array :: (Ix a) => (a,a) -> [(a,t)] -> (Array a t)
array  ebound
       ([(ie1,e1)  | gen-pred, ..]
     ++ [(ie2,e2)  | gen-pred, ..] ++ …)
```

Thus,

```
mkArray (l,u) f =
  array (l,u) [(j,(f j)) | j <- range(l,u)]
```

List comprehensions and function array provide flexibility in constructing arrays, and the compiler can implement them efficiently

duplicates?

Array Comprehension: *Wavefront*

x[i,j]= x[i-1,j] + x[i,j-1]

1	1	1	1	1	1	1	1
1							
1							
1							
1							
1							
1							
1							

```
x = array ((1,1),(n,n))
       ([((1,1), 1)]
     ++ [((i,1), 1) | i <- [2..n] ]
     ++ [((1,j), 1) | j <- [2..n] ]
     ++ [((i,j), x!(i-1,j) + x!(i,j-1))
                    | i <- [2..n],
                      j <- [2..n]   ])
```

Duplicates

- Haskell Semantics:
  - Enumerate the whole index range and return bottom if any duplicate indices are found
- pH Semantics:
  - Only the duplicated elements are bottom, no the whole array

- Haskell semantics are motivated by lazy evaluation and awful for parallel implementation; pH semantics are preferable if all the elements of the array are going to be computed

Another Issue: Computed Indices

Inverse permutation
y ! (x ! i) = i

2	5	6	1	3	4	x

⇓

4	1	5	6	2	3	y

```
find x i =
     let % find j such that x!j = i
          step j = if x!j== i then j
                                      else step j+1
     in
                step 1
y = mkArray (1,n) (find x)
```

How many comparisons? Can we do better?

```
y = array (1,n) [( x!i , i )| i <- [1..n]]
```

I-structures

In functional data structures, a *single construct* specifies:

- The *shape* of the data structure
- The value of its components

These two aspects are specified *separately* using I-structures

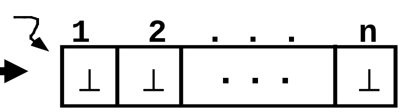
➔ efficiency
➔ parallelism

I-structures preserve *determinacy* but are *not* functional !

I-Arrays

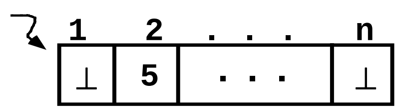
- *Allocation* expression

iArray (1,n) [] ➔



- *Assignment*

iAStore a 2 5
or **a!2 := 5**



provided the previous content was ⊥
"The single assignment restriction."

- *Selection* expression

a!2 ➔ **5**

(⊥ means empty)

Computed Indices Using I-structures

Inverse permutation
y ! (x ! i) = i

2	5	6	1	3	4	x

⇓

4	1	5	6	2	3	y

```
let
      y = iArray (1,n) []
      _ = for i <- [1..n] do
              _ = iAStore y (x!i) i
          finally ()    % unit data type
in
      y
```

More in a moment

What if x contains a duplicate ?

Multiple-Store Error

Multiple assignments to an iArray slot cause a multiple store error

A program with exposed store error is suppose to blow up!

Program --> T

The Top represents a contradiction

The Unit Type

```
data () = ()
```

means we cannot do much with an object of the unit type. However, it does allow us to drop '_ ='

```
let
   y = iArray (1,n) []
   for i <- [1..n] do
      iAStore y (x!i) i
   finally ()           --  unit data type
in
   y
```

For better syntax replace

`iAStore y (x!i) i` *by* `y!(x!i) := i`

I-Cell

```
data ICell a = ICell {contents :: . a}
```

I-Structure field

Constructor

```
ICell :: a -> ICell a

ICell e          or      ICell {contents = e}
```

or create an empty cell and fill it

```
ic = ICell {}
contents ic := e
```

Selector

```
contents ic      or
case ic of
     ICell x -> ... x ...
```

An Array of ICells

Example: Rearrange an array such that the negative numbers precede the positive numbers

2 8 -3 14 2 7 -5

-3 -5 2 8 14 2 7

Functional solutions are not efficient

```
let y = array (1,n) [(i,ICell {})| i<-[1..n]]
    (l,r) = (0,n+1)
     final_r = for j <- [1..n] do
                  (l',r',k) =
                        if (x!j >= 0)
                             then (l,r-1,r-1)
                             else (l+1,r,l+1)
                   contents (y!k) := x!j
                   next l = l'
                   next r = r'
                  finally r
in (y, final_r)
```


Type Issues

In the previous example

```
x :: Array Int
y :: Array (Icell Int)
```

1. IArray data type eliminates this extra level of indirection

2. The type of a functional array (Array) is different from the type of an IArray.

However, an IArray behaves like a functional Array after all its elements have been filled

We provide a primitive function for this conversion

```
cvt_IArray_to_Array ia -> a
```

Types Issue *(cont.)*

Hindley-Milner type system has to be extended to deal with I-structures

⇒ *ref type* -- requires new rules

don't have time to get into this ...

All functional data structures in pH are implemented as I-structures.

Array Comprehensions:
a packaging of I-structures

```
array dimension
      ([(ie1,e1) | x <- xs, y <- ys]
    ++ [(ie2,e2) | z <- zs] )
```

translated into

```
let   a = iArray dimension []
      for x <- xs do
            for y <- ys do
                  a!ie1 := e1
            finally ()
      finally ()
      for z <- zs do
            a!ie2 := e2
      finally ()
in  cvt_IArray_to_Array a
```

We have used pH syntax but it is trivial to translate this into Haskell syntax

I-structures are *non functional*

```
f x y = let x!1 := 10
            y!1 := 20
        in ()
```

```
let x = iArray (1,2) []
in  f x x
```

≡

```
f  (iArray (1,2) []) (iArray (1,2) []) ?
```

The example

```
f x y = let x!1 := 10
            y!1 := 20
        in ()
```

```
let
 x = iArray (1,2) []
in
 f x x
 ⇓
let
 x = iArray (1,2) []
 x!1 := 10
 x!1 := 20
 ⇓
 "blow up"
```

```
f  (iArray (1,2) [])
   (iArray (1,2) [])
⇓
let
 t1 = iArray (1,2) []
 t2 = iArray (1,2) []
 t1!1 := 10
 t2!1 := 20
in ()
```


We have finally slipped into
- parallelism issues
- side-effects

More on these issues after the quiz